This Page Is Inserted by IFW Operations
and is not a part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problems Mailbox.

Google Advanced Search Preferences Language Tools Search Tips

kernel level debugging Google Search

Web · Images · Groups · Directory · News ·

Searched the web for kernel level debugging. Results 11 - 20 of about 187,000. Search took 0.20 seconds.

Metrowerks News

Metrowerks Adds Linux® **Kernel-Level Debugging** Capabilities to CodeWarrior Development Environment. Metrowerks Adds Linux® **Kernel** ...
www.metrowerks.com/MW/news/default.asp?PR=475 - 17k - Cached - Similar pages

Sponsored Links

Kernel Debugging Help
OSR Open Systems Resources, Inc.
Debugging, Dump Analysis & Training
www.osr.com
Interest.

See your message here...

Electronic Design: Updated Tool Does **Kernel-Level Debugging** For ...

... Here: Articles > Electronic Design > Oct 2, 2000 > Article. Updated Tool Does **Kernel-Level Debugging** For Real-Time Linux.(Product Information) Electronic Design ...
www.findarticles.com/cf_dls/m3161/ 20_48/66219045/p1/article.jhtml - 12k - Cached - Similar pages

LWN: Metrowerks Adds Linux **Kernel-Level Debugging** Capabilities

To: <lwn@lwn.net>. Subject: Metrowerks Adds Linux **Kernel-Level Debugging** Capabilities. Date: Tue, 12 Aug 2003 08:56:07 -0500. Metrowerks ...
lwn.net/Articles/44131/?format=printable - 14k - Cached - Similar pages
[More results from lwn.net]

Comments for Upgraded Tool Does **Kernel-Level Debugging** For Real ...

[Reader Comments] Upgraded Tool Does **Kernel-Level Debugging** For Real-Time Linux, READER COMMENTS: We want to hear what you have to say about this article! ...
www.elecdesign.com/ Articles/Index.cfm?ArticleID=4759&Action=Comments - Similar pages
[More results from www.elecdesign.com]

kernel source-**level debugging**

... **kernel** source-**level debugging**. ... Hi, While searching for **kernel** source-**level debugging** on google, came across the following message. ...
www.mail-archive.com/kdb@oss.sgi.com/msg00687.html - 7k - Cached - Similar pages

Re: **kernel** source-**level debugging**

... Re: **kernel** source-**level debugging**. From: Keith Owens; Subject: Re: **kernel** source-**level debugging**; Date: Fri, 26 Sep 2003 06:48:23 -0700. ...
www.mail-archive.com/kdb@oss.sgi.com/msg00688.html - 6k - Cached - Similar pages
[More results from www.mail-archive.com]

kgdb: Source **level debugging** of linux **kernel**

kgdb: Source **level debugging** for linux **kernel**. What's New ... originally. He is the guy who started source **level debugging** on linux **kernel**. Thanks ...
www.jimbrooks.org/ web/linux/docs/kernel/kgdbSourceLevelDebuggingForLinuxKernel.php - 6k - Cached - Similar pages

Kernel level debugging

Kernel level debugging. ... Organization: TiemSys. Hi All, I am thinking of using insight to do some **kernel level debugging** of a remote host. ...
sources.redhat.com/ml/insight/2000-q2/msg00279.html - 4k - Cached - Similar pages
[More results from sources.redhat.com]

Linux-**Kernel** Archive: kdb source **level debugging**

... Adding source **level debugging** to kdb would allow me to understand the **kernel** source code much better, since i would be able to trace thru the code and see it ...
www.uwsg.iu.edu/hypermail/linux/ kernel/0211.0/1966.html - 5k - Cached - Similar pages

kgdb: Source **level debugging** of linux **kernel**
Kernel debugging: using gdb for **kernel debugging** Usage of gdb **kernel debugging** is similar to that for **debugging** application processes. ...
kgdb.sourceforge.net/gdbusage.html - 4k - Cached - Similar pages
[More results from kgdb.sourceforge.net]

◀ Goooooooooogle ▶
Result Page: **Previous** 1 2 3 4 5 6 7 8 9 10 11 **Next**

kernel level debugging Google Search Search within results

Google Home - Advertise with Us - Business Solutions - Services & Tools - Jobs, Press, & Help

©2004 Google

Google Advanced Search Preferences Language Tools Search Tips

breakpoint page inode linux Google Search

Web · Images · Groups · Directory · News ·

Searched the web for breakpoint page inode linux. Results 181 - 190 of about 790. Search took 1.29 seconds.

Using and Maintaining GNU Pascal
... list on the GPC home **page** in the ... fields `User' , `Group' , `Mode' , `Device' , `**INode**' , `SymLink' , `TextBinary ... the program with a **breakpoint** between statements ...
sci.univ-bpclermont.fr/logiciels/ scisrv/gpc/html/gpc.html - 101k - Cached - Similar pages

From rgooch@ras.ucalgary.ca Sun Apr 16 00:05:25 2000 Message-id ...
... has nothing that can be used for **inode** numbers that ... Next step if waiting on a **page** - ideas appreciated. ... the keyboard and waiting for the next >**breakpoint** to be ...
ftp.ussg.iu.edu/linux/mail.archive/ kernel/kernel.0004.2 - 101k - Cached - Similar pages
[More results from ftp.ussg.iu.edu]

This kgdb will get called and will trap almost any kernel fault ...
... * For a **page** fault, when we ... */ + **breakpoint**(); + } kgdb_gdb_message(s,count ... */ + +static int kgdb_consdev_open(struct **inode** * **inode**, struct file * file ...
www.bzimage.org/people/akpm/patches/2.5/2.5.67/ 2.5.67-mm3/broken-out/kgdb-ga.patch - 101k - Cached - Similar pages

<html> <head> </head><body><pre> This kgdb will get called and ...
... * For a **page** fault, when we ... */ + **breakpoint**(); + } kgdb_gdb_message(s, count ... */ + +static int +kgdb_consdev_open(struct **inode** *inode, struct file *file ...
www.bzimage.org/people/akpm/patches/2.6/2.6.0-test7/ 2.6.0-test7-mm1/broken-out/kgdb-ga.patch - 101k - Cached - Similar pages
[More results from www.bzimage.org]

[PS] DENX PPCBoot and **Linux** Guide (TQM8xxL) Wolfgang Denk wd@denx.de
File Format: Adobe PostScript - View as Text
... If you cannot find kermit on the distribution media for your **Linux** host system, you candownload it from the kermit project **page**: http://www.columbia.edu/kermit ...
www.denx.de/doc/TQM8xxL/dplg.ps - Similar pages

It's been nearly two years since the last revision, and quite a ...
... Do you have the **breakpoint** scheme that MACLISP is known for ... Filled it's data up with NULLs, cleared the **inode**, closed the ... I've lost my tty I wish my **page** hadn't ...
iimspc.its.monash.edu.au/nerdsongs.txt - 101k - Cached - Similar pages

Overkill - Bloody 2D action deathmatch-like game in ASCII-ART ...
... source navigation and lookup; **breakpoint**, watchpoint, backtrace ... including optimal paragraph and **page** breaking, automatic ... system – ie, on the active **inodes**. ...
ftp.tiscali.nl/conectiva/6.0/cd2/PACKAGES - 101k - Cached - Similar pages

Index: usr.bin/spell/special.4bsd ...
... Bourne box +Brauer brconfig break +**breakpoint** breaksw brk ... initgroups initstate +inline inmax +**inode** insque install ... 840,51 +1252,81 @@ packf **page** pagesize pagsh ...
www.davidkrause.com/ openbsd/patch-src_usr.bin_spell_special.4bsd - 17k - Cached - Similar pages

Set these options for all host adapters. * - Memory mapped IO does ...
... allow structures to cross **page** boundaries and ... d : kmalloc(%d) of **breakpoint** structure failed ... struct Scsi_Host * inode_to_host (struct **inode** *inode) {$ int dev ...

www.pell.portland.or.us/~orc/Code/linux-1.2.13/ drivers/scsi/53c7,8xx.c - 101k - Cached - Similar pages

[PDF] Writing Apache Modules with Perl and C Lincoln Stein Doug ...
File Format: PDF/Adobe Acrobat - View as HTML
... **Page** 15. 14 Installing /usr/local/lib/perl5/site_perl/i586- **linux**/./auto/MD5/MD5.so Installing /usr/local/lib/perl5/man/man3/./MD5.3 ... Writing /usr/local/lib ...
library.n0i.net/programming/perl/wr-apmod-pdf/ - Similar pages

◀ Goooooooooooooooooogle ▶
Result Page: **Previous** 9 10 11 12 13 14 15 16 17 18 **19** 20 21 22 23 24 25 26 27 **Next**

breakpoint page inode linux Google Search Search within results

Google Home - Advertise with Us - Business Solutions - Services & Tools - Jobs, Press, & Help

©2004 Google

US PATENT & TRADEMARK OFFICE
3 0402 00270 5079
EIC 2100

How Debuggers Work

Algorithms, Data Structures, and Architecture

Jonathan B. Rosenberg

Software Development / Testing & Quality

$50.00 USA
$62.95 CAN

A total guide to debuggers: what they do, how they work, and how to use them to produce better programs

"Debuggers are the magnifying glass, the microscope, the logic analyzer, the profiler, and the browser with which a program can be examined."

—*Jonathan B. Rosenberg*

Debuggers are an indispensable tool in the development process. In fact, during the course of the average software project, more hours are spent debugging software than in compiling code. Yet, too many programmers really know how to constructively interpret the results they get back from debuggers. And even fewer know what makes these complex suites of algorithms and data structures tick. Now in this extremely accessible guide, Jonathan B. Rosenberg demystifies debuggers for programmers and shows them how to make better use of debuggers in their next projects.

Taking a hands-on, problem-solving approach to a complex subject, Rosenberg explains how debuggers work and why programmers use them. Most importantly, he provides practical discussions of debugger algorithms and procedures for their use, accompanied by many practical examples. The author also features a wide variety of systems applications, from Microsoft's Win32 debug API to a large parallel architecture.

Visit our Web site at
http://www.wiley.com/compbooks/

JONATHAN B. ROSENBERG, author, manages the development of software tools at Borland International.

WILEY COMPUTER PUBLISHING
John Wiley & Sons, Inc.
Professional, Reference and Trade Group
605 Third Avenue, New York, NY 10158-0012
New York • Chichester • Brisbane • Toronto
Singapore • Weinheim

ISBN 0-471-14966-7

55000
9 780471 149668

Chapter 6

Breakpoints and Single Stepping

Breakpoints

Breakpoints are key to all debugger execution control—almost all execution control algorithms, at some point, involve breakpoints. These algorithms frequently require a special breakpoint be set, perhaps completely invisible to the user. This section describes a set of requirements for breakpoint algorithms, data structures for breakpoints, and various scenarios and algorithms that may be used by debuggers to fulfill the requirements stated.

Requirements for Breakpoint Algorithms

Following is a list of requirements for a debugger's basic breakpoint mechanisms. Breakpoints must adequately support execution control functionality and provide the rich set of functionality nedeed for a modern debugger:

- User may insert source-level or instruction level breakpoints.

- Some high-level breakpoints may map onto many text addresses such as breakpoints in C++ templates.
- Each user-created breakpoint must be represented and maintained uniquely.
- There can be many user-created breakpoints at the same source code or text address location.
- Some user-created breakpoints will be temporary ("once only") and must be removed at the next stop.
- Breakpoints may have associated conditions that must be evaluated by the debugger to determine if the stop really should occur.
- Breakpoints may have side effects that must be acted on by the debugger when activated but that may or may not actually be represented to the user as a debuggee stop.
- Breakpoints set in not-yet-loaded modules must be resolved when these modules get loaded.
- Internal breakpoints created by the debugger must be maintained so that they are invisible to the user.
- On multiprocessor architectures, high-level breakpoints may require interprocessor synchronization.

Breakpoint Data Structures

Typically there need to be at least two "levels" of breakpoint representation: the logical and the physical. The **logical breakpoints**—usually corresponding to those set by the user—are those associated with a point in the source code. The **physical breakpoints**—those that relate directly to executable machine instructions—are the points in the text space where actual hardware breakpoint instructions get written. It is the physical level that must store the original instruction (or part thereof) that must be replaced if the breakpoint is to be removed. The logical level is responsible for representing a breakpoint as fully resolved (that is, it has a mapping to a physical address) or as not yet resolved as when a breakpoint is set in a module that will not be loaded until some time later during debuggee execution. Special kinds of breakpoints called "conditional breakpoints" may or may not actually stop when the breakpoint "fires" depending on the value of an associated condition.

These conditions associated with a breakpoint are maintained at the logical level. This is true even for logical breakpoints set in disassembled instructions presented to the user in a machine or CPU view. Conditions are Boolean expressions that are evaluated by the debugger upon breakpoint activation. If the Boolean expression evaluates to false, the breakpoint activation is ignored and execution automatically resumes without the user ever being notified of the stop. If the Boolean expression evaluates to true, the breakpoint activates and the user is notified the process has stopped. Such conditions as *pass counts*, *window message received*, and *expressions* that must evaluate to true for valid stop are also maintained at the logical level.

A many-to-one relationship may exist between logical and physical breakpoints, as shown in Figure 6.1. We do not restrict the user from setting two distinct breakpoints (perhaps with different conditions) at a point in the source code that maps to the same physical location. In fact, as we shall explore later, C++ templates cause the logical to physical mapping to be many-to-many. Because we expect the number of breakpoints to remain relatively small, the most effective approach usually employed is to have two separate structures for logical and physical breakpoints where the node in each list maintains an address that is the link between the two. The downward mapping from logical to physical occurs when setting, deleting, or modifying a breakpoint. This downward mapping results in a physical address to be used as a lookup or search token in the physical breakpoint list. The upward mapping occurs when a breakpoint triggers due to the debuggee executing a breakpoint instruction. This upward mapping results in a physical address from the OS that maps uniquely to one node in the physical breakpoint list. This same address is then used to search the logical breakpoint list to find all logical breakpoints that mapped into the given physical address. At this point, any associated conditions can be evaluated to determine if this stop of the debuggee should be reported to the user because some user-created breakpoint met all its conditions for stopping the debuggee. Because of the inherent many-to-one mapping between logical and physical, it is necessary for the physical level to know when all logical breakpoints referring to a single physical address are deleted or disabled. This is easily accomplished with reference counts on the nodes in the physical breakpoint list. A node is actually deleted, removed from the list, and its original instruction restored only when its reference count drops to zero.

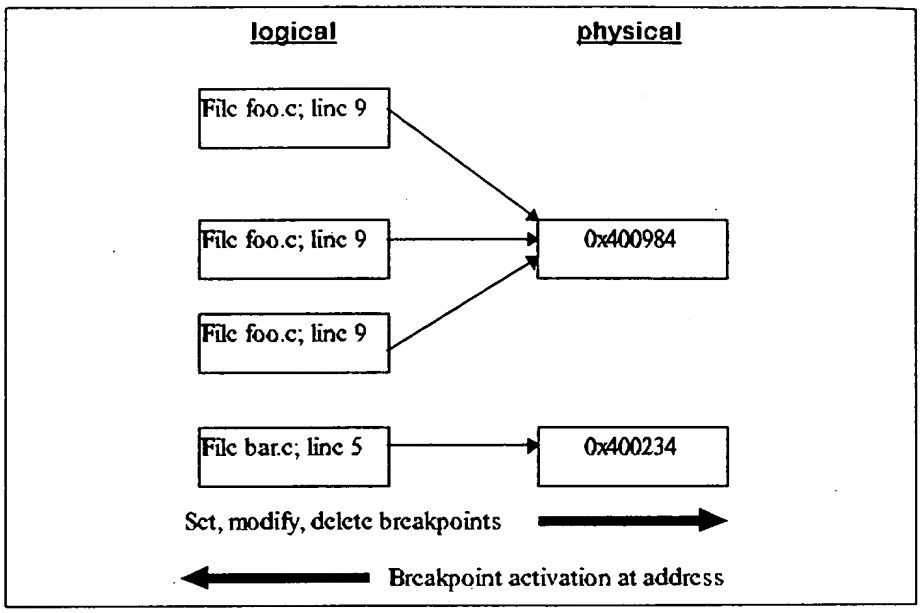


Figure 6.1

Breakpoint two-way mappings. *This figure shows that in some cases, several logical breakpoints map onto a single physical breakpoint location. The mappings are used in both directions. Logical-to-physical on set, modify or deletion of breakpoints. Physical-to-logical whenever a hardware breakpoint event occurs.*

Breakpoint Setting and Activation

The basic breakpoint setting algorithm, based on the data structures described above, is shown in Algorithm 6.1. The user specifies that a source line in the editor or source view should have a breakpoint set on it. This algorithm is then run to map that to a physical breakpoint location in the code space.

Algorithm 6.1 *Breakpoint setting (source level)*

Input File name and line number or file offset in source file.
Output Physical location of breakpoint or error indication.
Method Map from file name plus line number to physical address using symbol table, logical breakpoint, and physical breakpoint.

i. Request symbol table agent map given file name and line number information into physical address (notify if given module not yet loaded);

ii. Create logical breakpoint object with this information contained;
iii. Create (or increment reference count if already exists) physical breakpoint object;
iv. Physical breakpoint agent must now insert breakpoint instruction and save original instruction at that location;

The physical breakpoint object is very simple and looks like Figure 6.2. These structures are probably kept in a linked list because there are never so many of them that the linked list overhead becomes an issue.

There are two sides to this coin: setting the breakpoint and then hitting it once the process starts executing. Algorithm 6.2 shows the steps necessary once a breakpoint fires.

Breakpoint Validation

When a breakpoint is set by a user there may not be an address at which to physically place the breakpoint instruction yet. This can occur because there may not yet be a valid mapping from the source code the user can see and manipulate and the executable code, which executes on the processor. The breakpoint may be in code in a dynamically loaded library (DLL) that has not yet been loaded. Or, perhaps no process is loaded at all because the user is modifying the source code and the compiler has not yet translated the source code into executable text. In either case, the breakpoint set by the user, a logical breakpoint, will exist without any associated physical breakpoint until some later time—it remains invalidated. In fact, it may never be a valid breakpoint because it may have been set on a non-executable statement. Unless the editor parses the text being edited, it cannot know, until a statement table is built by the compiler, which places in the source text represent breakpointable locations. Once the process finally gets loaded or the appropriate DLL is loaded, a validation algorithm must complete the mapping of invalidated logical breakpoint to physical.

Validation of breakpoints must get triggered at the earliest possible moment. This is easy in the case of invalidated breakpoints in a process not yet loaded as a debuggee. When the process is first created no instructions are executed before the debugger has a chance to process all the invalidated breakpoints and get physical breakpoints inserted. It is slightly more problematic for DLLs loaded at run-time. More recent operating systems have provided a debugging *event* or notification when a module load or unload occurs. This

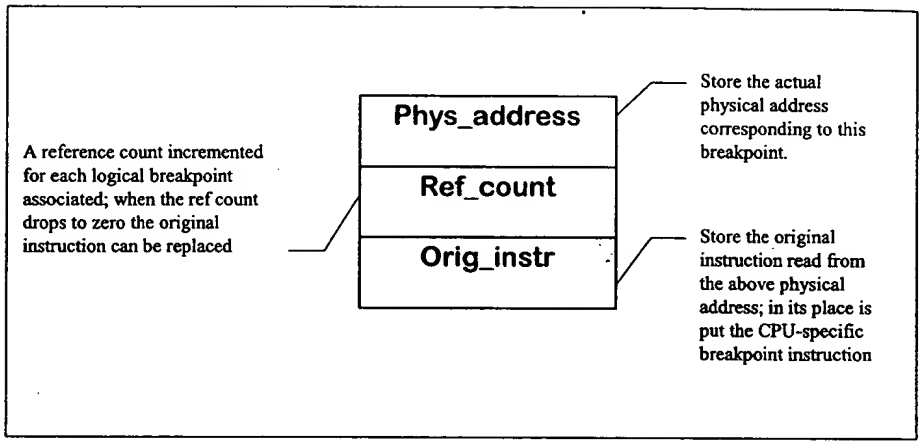


Figure 6.2

Simple physical breakpoint structure. *This structure holds the physical address in instruction memory where the breakpoint instruction will be placed. The original instruction is saved so that when the breakpoint must be removed the correct original instruction can be replaced. There is also a reference count due to the fact that multiple logical breakpoints may map to the same physical location. This allows the debugger to easily maintain the breakpoint and this corresponding data structure until the ref_count drops to zero.*

notification causes the debuggee to stop execution whenever a DLL is loaded but before any instruction in that DLL actually executes. This gives the debugger a chance to validate these DLL breakpoints in time to catch any breakpoints that might get hit as the DLL executes. Provision needs to be made within the debugger's execution control algorithm to process this kind of notification and validate all breakpoints in the source code associated with a DLL. The same procedure will have to run on initial startup for statically loaded libraries and on attach, where all previously loaded libraries will generate a series of module load events.

Older operating systems (like Windows 3.1 or before) required extensive trickery to accomplish this—although they support DLLs there was no debugger notification of these modules being loaded. Catching the call to the run-time routine (LoadLibrary()) that causes these loads was necessary. Debuggers would have to find this routine in the operating system's list of external entry points at debuggee startup time, set a special breakpoint in this routine that would trigger whenever a load module was requested, and handle this breakpoint being hit as if it were a module load notification. Algo-

Algorithm 6.2 *Breakpoint activation*

Input	OS notification that debuggee stopped at certain address due to breakpoint.
Output	Stop or continue debuggee according to below method.
Method	Map from physical location back to source file and line number using physical and logical breakpoint structures.

i. Scan logical list of breakpoints for this address (there may be more than one);
ii. Apply any conditions associated with this breakpoint (this may even involve more execution if the condition involves expression evaluation);
iii. If a condition does not evaluate to true move on to the next item in the list, ignoring this one any further;
iv. When no more breakpoints at this address are found, determine if any had either no conditions or had conditions that evaluated to true;
v. If so we report stop;
vi. Else we continue the debuggee as if no stop occurred at all.

rithm 6.3, to perform breakpoint validation, should be used whenever a debuggee process is created and at each subsequent module load event during debugging. The unverified list may indeed not be empty at the end of this process. This is because a file containing

```
1: #ifdef _IN_EXE
2: I=Ø;
3: #else //_IN_DLL
4: I=1;
5: #endif
```

is compiled into both an .EXE and DLL, a breakpoint on line 4 will show up as invalid in the EXE but can later be verified when the DLL is loaded.

Temporary Breakpoints

Breakpoints can have numerous attributes associated with them. They can be valid or invalid, as we have seen. They can be temporary or permanent. Temporary breakpoints—sometimes thought of as firing "once only"—are used to implement features such as "run-to-main" or "run-to-here." Run-to-main

Algorithm 6.3 *Breakpoint validation*

Input OS notification that a module load has occurred.
Output Processing of breakpoint list validating as many as possible.
Method As processes are created and when modules are loaded, find all unverified breakpoints and remap them from logical to physical.

i. Get module name being loaded either executable or DLL;
ii. Determine list of source files used to build this module from the compiler-generated symbol table modules section;
iii. Find all not-yet-validated breakpoints that match a file name from this list;
iv. For any matches, use the line number in the breakpoint object to lookup in statement tables a breakpoint address;
v. Report failure on those breakpoints on line numbers not in the list;
vi. Set physical breakpoints in executable code and mark breakpoint as validated;
vii. Continue with step iv above until all list entries have been examined.

is used at debuggee startup to quickly execute past all startup code and to stop on a program's main routine—logically considered by the programmer to be where the program starts.[1] Run-to-here allows the user to point to source code where he or she desires the program counter to be and quickly have the debuggee execute up to that point. These and other examples of temporary breakpoints are a convenient way to move execution to a certain point but do not require explicit setting and unsetting of breakpoints by the user. The debugger handles this invisibly. A temporary breakpoint is set, the debuggee is started running, and once it stops the temporary breakpoint removed. Typically, the temporary nature of the breakpoint is one of its attributes. In other respects it is just like all other breakpoints. Thus, the algorithm for cleanup of temporary breakpoints just requires a scan of all breakpoints whenever the debuggee stops, wherein the normal algorithm for breakpoints is applied to each one.

Internal Breakpoints

Like temporary breakpoints, internal breakpoints can be just an attribute associated with a breakpoint object. Internal breakpoints are invisible to the

[1]Erroneously, however, because C++ static constructors are executed before main and can be a problematical and error-prone area. Even in C, if # pragma startup is used the exact same problem occurs.

user but are key to the debugger's correct handling of many of its algorithms. These are breakpoints set by the debugger itself for its own purposes. I will discus several situations where internal breakpoints are used.

The basic source-level single-step algorithm uses a combination of internal breakpoints and full-speed run to key internal breakpoints placed for optimal stepping speed, as compared to machine-stepping instruction-by-instruction through hundreds or thousands of instructions. This is especially critical on source statement *step over*. Step over is a statement step in the context of the current function scope running any descendent functions full speed to completion. For statement step over, we typically use an internal breakpoint on a function's return address to allow that function and all its descendents to run at full speed. *Step into* is a statement step that goes into any descendent functions found during the current operation. Even step into uses internal breakpoints to quickly run over stretches of code as long as possible up to some branch instruction, again for performance reasons. Both of these algorithms will be described in much more detail later in this chapter.

Some processors, especially the newer generations of very high-performance RISC processors, do not provide any hardware single-step support because it can complicate a processor that is focused on simplicity and performance. In this case, the debugger has no choice but to set internal breakpoints and run to the next breakpoint to simulate the single-step functionality. To do this correctly, the debugger must decode the current, about-to-be-executed instruction. If the next instruction is non-branching, the debugger may set the breakpoint just past that instruction and run. If the debugger detects the next instruction as a branch instruction it must decode the target of the branch and set a breakpoint there to correctly "single-step" over the branch instruction. Or the debugger could trace to both possible targets of the branch instruction and set breakpoints at both addresses to avoid the prediction of the branch target.

During expression evaluation, when the debugger must use the debuggee to evaluate a function in an expression entered by the user, internal breakpoints are used to carefully control execution so that just the desired function is executed. An internal breakpoint must be placed at the return address of the function being called so that once evaluation completes, the debuggee stops to allow complete cleanup so that normal debuggee execution can proceed later. More details on the expression evaluation algorithm appear in Chapter 8.

If a very different model of stepping is required, internal breakpoints may be employed to accomplish this different approach. For example, where a program consists entirely of a collection of disjoint user-written, event-driven small functions, stepping off the end of one of these functions may mean the debugger must run the program to the beginning of the next user-written function. This would require a special internal breakpoint to catch the run from the end of one function to the beginning of the next function.

One additional consideration about internal breakpoints is critical in debugger design. This is the question of when (if ever) these breakpoints are visible to the user. In a disassembly view, where machine instructions are disassembled into their mnemonic equivalents, we may want to ignore the existence of internal breakpoints and show the underlying instruction instead. But a hex memory dump may want to show the exact contents of memory even if this includes the debugger-inserted internal breakpoints.

Side Effects

Breakpoints can be used for much more than just a way to stop the debuggee program and give control to the user. Side effects on breakpoints allow a lot of interesting debugging approaches. It is perhaps best to think of breakpoints as probe points where test data can be extracted as the program runs and does not necessarily stop. We can program the debugger to do anything we deem useful when a breakpoint activates by associating *actions* to be performed when a breakpoint evaluates its condition (if any) to be true.

Logging is a simple action that can be performed at a breakpoint. Frequently all that is desired is a record of the activation of a breakpoint in some sort of historical readout. All the debugger is directed to do is to emit some characteristic information about the breakpoint that can be collected for the user to review. The debugger may or may not be directed to stop at this particular breakpoint. This is very similar to a debugging technique familiar to all programmers: inserting print statements directly in the source code that record some sort of history of execution when the program runs. However, the breakpoint history record approach via a debugger has the advantage of not requiring the program to be recompiled.

Pass counts are another frequently used side effect. Pass counts are simple expressions to be evaluated by the debugger upon breakpoint activation. The

If a very different model of stepping is required, internal breakpoints may be employed to accomplish this different approach. For example, where a program consists entirely of a collection of disjoint user-written, event-driven small functions, stepping off the end of one of these functions may mean the debugger must run the program to the beginning of the next user-written function. This would require a special internal breakpoint to catch the run from the end of one function to the beginning of the next function.

One additional consideration about internal breakpoints is critical in debugger design. This is the question of when (if ever) these breakpoints are visible to the user. In a disassembly view, where machine instructions are disassembled into their mnemonic equivalents, we may want to ignore the existence of internal breakpoints and show the underlying instruction instead. But a hex memory dump may want to show the exact contents of memory even if this includes the debugger-inserted internal breakpoints.

Side Effects

Breakpoints can be used for much more than just a way to stop the debuggee program and give control to the user. Side effects on breakpoints allow a lot of interesting debugging approaches. It is perhaps best to think of breakpoints as probe points where test data can be extracted as the program runs and does not necessarily stop. We can program the debugger to do anything we deem useful when a breakpoint activates by associating *actions* to be performed when a breakpoint evaluates its condition (if any) to be true.

Logging is a simple action that can be performed at a breakpoint. Frequently all that is desired is a record of the activation of a breakpoint in some sort of historical readout. All the debugger is directed to do is to emit some characteristic information about the breakpoint that can be collected for the user to review. The debugger may or may not be directed to stop at this particular breakpoint. This is very similar to a debugging technique familiar to all programmers: inserting print statements directly in the source code that record some sort of history of execution when the program runs. However, the breakpoint history record approach via a debugger has the advantage of not requiring the program to be recompiled.

Pass counts are another frequently used side effect. Pass counts are simple expressions to be evaluated by the debugger upon breakpoint activation. The

debugger must record the number of times this particular breakpoint has activated. This information is compared against a *threshold*—the pass count—specified by the debugger user. This is a common way to control a certain number of passes through a programmed loop before execution of the debuggee stops. This is for bugs that follow the pattern "it happens after N times through the loop."

A more general form of expression evaluation than pass counts is possible and common as well. Everything from specialized expression languages using debugger-created variables, to the full expression syntax of the debuggee's programming language using debuggee-based variables, is possible. This can be an extremely complex and involved portion of the debugger, and it is covered in detail in Chapter 8. Here it suffices to say that as with other breakpoint side effects, on breakpoint activation the expression may be evaluated. If this is a conditional breakpoint, then the expression is evaluated for its Boolean value. If true, the breakpoint will cause a stop; otherwise, execution will continue. Or, the expression may be tied to the logging feature and the value of the expression recorded in the viewable log. These uses of expression evaluation are a way to have the debugger "patch" the debuggee program without modifying the program or even compiling it at all. In the first case, an expression evaluated at breakpoint activation may itself have side effects that "fix" some problem or deficiency in the program being debugged. For example, the expression may set a variable to zero that was previously uninitialized in the program that seems to fix a problem. This "fix" can now be tested and verified before modifying the program itself. The logging expression case is like having added a print statement to the program and recompiled it except not as fast or in as flexible a manner as having the debugger do it.

Other side effects are possible as well. For instance, it is useful in message-based GUI program debugging to carefully track the GUI messages received at a specific function. Specialized breakpoint side effects can be created that track the GUI messages being processed when the breakpoint activates. Then either evaluating this in a conditional sense to determine if a stop should occur or simply logging the receipt of that message and proceeding is possible.

Finally, arbitrary "actions" may be associated with a breakpoint as a further side effect. These actions can consist of any of the capabilities of the debugger that can be expressed in some type of macro language as a single or

linked set of functions. For example, setting another breakpoint may be the desired action when a special breakpoint activates. The trend is toward more and more configurable tools that can be driven from a macro or embedded programming language. Once a debugger can be driven in this fashion any of its capabilities can then be associated as an action.

C++ Templates and One-to-Many Problems

Breakpoints set in source code that is then replicated by the compiler pose special problems. This scenario creates a one-to-many mapping from source code to executable. Breakpoints in inlined code, some breakpoints on function returns (depending on the compiler), and some breakpoints on for-loops (again, depending on the compiler) all share this characteristic. However, the problem is extreme in C++ templates, and the debugger design must handle this extreme case smoothly as C++ templates are becoming very prevalent.

As the breakpoint setting algorithm looks up a file name / line number pair it may find that many executable modules have this mapping and that many physical addresses correspond for locating the physical breakpoint instruction. At this point the debugger may opt to give the user a chance to filter this down to select just the ones intended to receive the breakpoint. This may work well for C++ templates because each of the C++ templates represents an instantiation for a distinct type and the user may be thinking of only one of these types as he or she sets the breakpoint.

Code Patching by the Debugger

Breakpoints are also the basis for more extreme code modifications attempted by some debuggers. Instead of inserting a special breakpoint instruction at a given location and saving the original instruction away in debugger memory, any instruction could be inserted into the executable code stream by the debugger. Specifically, a branch or jump instruction could be inserted. What this means is that very general code patching is possible. Suppose, for example, a special monitor routine needs to be called upon entry to every function. The debugger could place this monitor routine in code memory and then insert special stack manipulation and jump instructions to cause in-line redirection to this monitor routine. This allows us to make these significant changes to the debuggee program without a recompile. There are

lots of uses for code patching in debuggers and other tools related to debuggers such as profilers.

Using standard debugging techniques the user may have a theory about what will fix the bug, but rebuilding the entire program may take so long that some faster way to verify if the proposed fix works is frequently required. One way to offer this capability is to allow users to modify the debuggee program with new and/or different code without re-compilation. The most common way to do this is through the function evaluation capability described in Chapter 8. Used in conjunction with a breakpoint that allows an associated expression evaluation, this kind of code patching support is straightforward. As long as breakpoints have an option that allows the user to specify that execution does not actually stop but some expression will get evaluated instead, and as long as general expression evaluation includes debuggee function invocation, the desired code patching capability is provided as desired. Further refinement that provides one more level of support for incremental code modification is to allow these function invocations to support breakpoints. This extra level of generality allows a rich set of features to be offered to the debugger user, which may lead to powerful debugging support for difficult debugging problems even in large, complex programs.

Single-step

Single-step is important because users need to be able to "watch" execution proceed. Frequently the failure mechanism, to be understood, must be "eased into." It is also important to understand the side effects occurring close to the failure point, which stepping allows. To fully support breakpoints and single-step in its various forms, the debugger needs to have a very sophisticated execution control mechanism. The main reason for this is the asynchronous nature of debug event notifications that occur whenever the debuggee stops. It is not correct to start a single-step assuming the next stop must be the completion of the just initiated single-step—it could stop instead for a breakpoint or exception. This requires a fairly involved set of states maintained by the debugger that correctly describes the state of the debuggee with respect to its current execution algorithm. Motivation for this statement comes easily with a simple example. During statement step, which requires many internal breakpoints and instruction-level steps, a divide-by-zero occurs, but the sin-

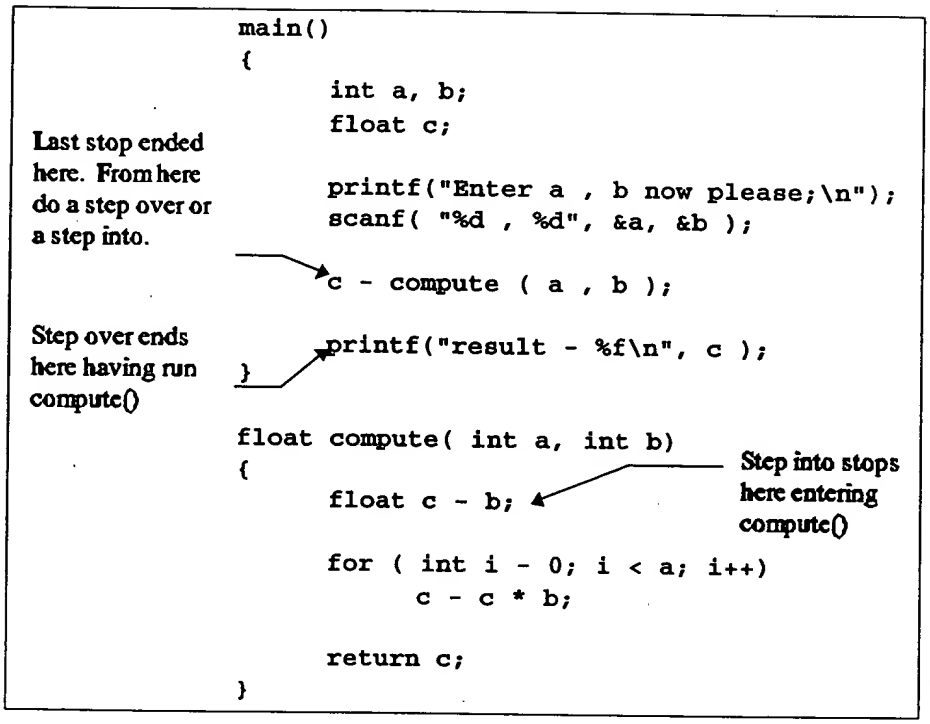


Figure 6.3

Sample code showing step over versus step into. *This figure shows sample code to explain the difference between step over and step into. When the program is stopped on a statement containing a call to a function, step into visits the function while step over stops on the line after the function returns.*

gle-step initiation is forgotten as less important than the divide-by-zero. The debugger must report the exception if this type of exception is supposed to be reported and then stop after cleaning up all the state being maintained for the statement step. Typically this is best handled by a finite state machine that gets its initial state from the user-requested command. Each notification from the OS about the debuggee stopping is a transition in this finite state machine, possibly to a new state. Breakpoint notifications occur at several points in the state diagram for this finite state machine. If the debugger was executing a statement step and this was an internal breakpoint, then the debugger must determine if the current location corresponds to a valid

source statement boundary that represents the completion of one source step. If not, execution proceeds either to another internal breakpoint or via instruction level single-step.

Step Into versus Step Over

We define "step into" to mean execution proceeds *into* any function in the current source statement and stops at the first executable source line in that function. "Step over," sometimes referred to as "skip" instead of step,[2] treats a call to a function as an atomic operation and proceeds past any function calls to the textually succeeding source line in the current scope. Figure 6.3 shows the distinction on a simple code fragment.

Step into could be implemented by using machine step repeatedly checking at each instruction step to see if the current address matches a source statement's starting address. In practice it is a serious performance problem to do this. This fact, plus the existence of processors that do not support machine-level step, leads us to an algorithm that decodes instructions and advances the processor via breakpoint plus full-speed run. More on this later in this chapter, where we discuss smart, fast stepping algorithms. First, here is Algorithm 6.4, the basic algorithm for stepping.

Algorithm 6.4 *Source step into*

Input Current statement and instruction pointer address.
Output A new statement and instruction pointer address for the next statement to be executed.
Method

i. Note current statement location;
set moved_flag := false;
ii. On each debug stop notification (and after initial setup):
iii. Set simulated_pc := real_pc;
iv. If (moved_flag == true and simulated_pc points to beginning of a statement) then
step into is completed so report new location to the user and exit this algorithm;
else if (no source avail) we can either run or "run to first source" (see page 128)

[2]Step over, aka skip, is called "next" in some older debuggers such as some variants of UNIX's dbx.

v. Get and decode the instruction pointed to by the simulated_pc in debuggee text address space;
vi. If (this instruction is not any sort of branching instruction and it is not an exact match for the beginning of a source statement) then
advance our simulated_pc to the next instruction;
go back to step iii;
vii. If (simulated_pc does match beginning of a new source statement and real_pc != simulated_pc) then
set a temporary breakpoint here;
set the moved_flag := true;
set the debuggee running;
wait for the next debug notification;
clean up internal breakpoint just hit;
go to step iii;
viii. If (this is a branching instruction) then
either
machine single-step one instruction;
or
decode the branch target;
set a temporary breakpoint at that address;
set the moved_flag := true;
start the debuggee running;
wait for the next debug notification;
clean up internal breakpoint just hit (if any);
go to step iii;

The algorithm for step over begins just like step into but it notes a "call" instruction during instruction decode as special, and once the called function has been entered, a breakpoint is inserted at the return address of this function by looking at the current stack frame. A full-speed run then gets the debuggee rapidly through this function and any functions it calls (assuming no breakpoints or exceptions are encountered along the way). Once this return address breakpoint is reached, after breakpoint cleanup, the basic source step algorithm is continued. The result will be to skip over a function and to step to the textually next source statement within the current function scope. Algorithm 6.5 for step over is shown next.

Algorithm 6.5 *Source step over*

Input Current statement and instruction pointer address.
Output A new statement and instruction pointer address for the textually 'next' statement.
Method

i. Note current statement location;
set moved_flag := false;
set in_function flag := false;
ii. On each debug stop notification (and after initial setup):
iii. Set simulated_pc := real_pc;
iv. If (moved_flag == true and in_function flag == false and simulated_pc points to beginning of a statement) then
step over is completed so report new location to the user and exit this algorithm;
v. If (in_function == true) then
use current stack frame to find return address for the new current function;
set internal breakpoint at this return address;
set in_function := false;
set debuggee running;
wait for debug notification;
go to step iii;
vi. Get and decode the instruction pointed to by the simulated_pc in debuggee text address space;
vii. If (this instruction is not any sort of branching instruction and it is not an exact match for the beginning of a source statement) then
advance our simulated_pc to the next instruction;
go back to step iv;
viii. If (this instruction is a function call instruction) then
set in_function := true;
machine single-step into this function;
wait for debug notification;
go to step iii;
ix. If (simulated_pc does match beginning of a new source statement and real_pc != simulated_pc) then
set a temporary breakpoint here;
set the moved_flag := true;

```
                set the debuggee running;
                wait for the next debug notification;
                clean up internal breakpoint just hit;
                go to step iii;
        x.  If (this is a branching instruction) then
                either
                        machine single-step one instruction;
                or
                        decode the branch target;
                        set a temporary breakpoint at that address;
                set the moved_flag to true;
                start the debuggee running;
                wait for the next debug notification;
                clean up internal breakpoint just hit (if any);
                go to step i;
```

Smart, Fast Source-Step

If a debugger implements source-step by using a series of machine single-steps, checking the text address reached each time against the statement address table, single-step will be painfully slow at times and users will not tolerate it. Typically, machine single-step is roughly 1000 times slower than full-speed run to a breakpoint. There is a tremendous cost to a call to the OS debug API due to several context switches (debugger to OS, OS to debuggee, debuggee to OS, OS back to debugger) and OS scheduling delays as well as CPU overhead. Second, the number of instructions to execute may be large. If there is a call to a library routine such as `printf` that does not have associated source code to stop in and show the user, thousands of individual calls to machine single-step would be required

This is why Algorithms 6.4 and 6.5 used instruction decoding as a fundamental part of the algorithm, as opposed to multitudes of instruction steps. Skipping over entire functions by setting a breakpoint at the return address and running full speed to this breakpoint dramatically decreases the number of single steps attempted, which in turn dramatically speeds up source step.

Instruction decoding involves reading a debuggee text address (at the current program counter usually) and applying a CPU-specific lookup to determine the type of instruction at this location. If the instruction is a procedure call instruction, the debugger knows one machine step will execute that instruction and

end up at the first instruction inside that function. Now, typically obtaining the value in a special register gives the debugger the return address where a breakpoint can be set to enable skipping over the entire function rapidly.

More extensive instruction decoding can be used to decrease usage of machine single stepping even more. Sequences of in-line, that is, non-branching, instructions can be grouped together and executed all at once by setting a breakpoint at the end of such a block of instructions. If these sequences tend to be long, this approach can save a large percentage of calls to the OS debug API. This level of instruction decoding just requires detecting branching instructions versus non-branching instructions.

This can be taken a step further by decoding targets of branches and setting a breakpoint only when an instruction decode requires dynamic data or a statement boundary has been reached. In fact, this approach is required if the processor does not support machine single-step. And even on processors that do support it, it may be faster not to use it and to fully decode instructions always running full speed to the next breakpoint.

It is possible to go even further and completely emulate each instruction (for those instructions where this is even possible) so that even data dependent branches can be decoded correctly. In most cases this would allow a source statement to be "executed" without ever running the CPU. It is not clear whether this is important enough to justify the significant extra logic in a debugger. One last important point about instruction decoding in debuggers is that this area is one of only three in a typical debugger that are processor-specific and non-portable. The three areas that are processor-specific and non-portable are the following:

1. Instruction decoding as used in stepping algorithms
2. Stack back-trace unwinding or "walking"
3. Disassembly and CPU view register presentation

It is worth keeping these aresa isolated from all other debugger functions to enable easier porting to new processors.

Pathologic Stepping Problems

Nothing is as simple as being able to uniformly apply the above simple algorithms and get the correct behavior out of single-step in all cases. Many

contemporary debuggers have anomalous stepping behavior under some circumstances. Two examples of commonly found anomalous behavior are single-*line for loops* and attempting to step into "missing" routines. Because the debugger's basic approach to stepping is so simple, it can be argued that these stepping problems stem from incomplete or inconsistent information provided by the compiler. We will examine these issues in more detail in the following sections.

Single-line For Loops

Many C/C++ compilers given the source

```
for ( i = 0; i < 1000; i++ )
    a += i;
```

generate this code (x86 variant shown):

```
    xor     eax, eax
@2:
    add     edx, eax
    inc     eax
    cmp     eax, 1000
    jl      short @2
```

and will generate only one breakpointable address for this single syntactic language statement (even though it resides on two textually distinct lines). This means that a user single-stepping from the beginning of the for-loop will see the debuggee advance all the way through the for-loop, stopping on the next line textually after the for-loop. The user might be surprised by this result because if he or she had written

```
for ( i = 0; i < 1000; i++ ) {
    a += i;
    b *= i;
}
```

stepping will naturally step to each line inside the {} 1000 times.

Debugger stepping algorithms need to be prepared to deal with this single-line for-loop situation. If the compiler does not help the debugger, the debugger can still behave correctly. A simple-minded approach would be to notice if the CPU branch instruction within a single-source statement branches backward to an address still within the same statement. Now the debugger considers this branch instruction as a stoppable location (even though it does

not match the beginning of a source statement, as expected by our source-stepping algorithm). This solution is not foolproof because compilers are not prevented from generating backward branches within a statement.

Similar to the single-statement for-loop is multiple return instructions from within a single function. In this case there is no one-to-one mapping between source and statement line number tables. This situation has many breakpointable locations for a single source line (the function's closing }), whereas the for-loop example had several source lines and only one breakpointable location. The debugger must be able to deal with both types of scenarios.

Step Into "Missing" User Routines

Source-level single step should present the "illusion" that the high-level language *is* being executed directly, one statement at a time. But sometimes, the reality of how a debugger implements single-step comes in direct conflict with this illusion. Then, something has to give. Either the user will be suprised—usually not in a positive way—or the debugger will have to do something extraordinary to satisfy this conflict.

One example of this is the classic dilemma of source stepping through a mixture of functions with debug information (the ones supplied by the user) with functions without debug information (as supplied by a run-time library, for example). If a user routine calls a library function that in turn calls a user routine (a "callback"), the standard step algorithm will not give the desired results. The "hidden" user routine—the one called by the immediately contained library (no debug) routine—will never be seen by source step. This is not a contrived example. It is now extremely common, especially in event-driven systems like Microsoft Windows and UNIX X-Windows, because a user supplies user routine pointers to the basic event-processing loop. Top-level user code calls the library event-processing routine, which on some events calls these user-supplied dispatch routines. Because single-step *into* promised to take you to the next executed source statement that has debug information, it has broken its promise. Instead, as step into detects it has entered a routine—the one in the library—it sets an internal breakpoint at this function's return address and runs full speed over this function and all the functions it calls. It also just ran over the user function called from inside the library routine. This is a serious flaw in single-step and one not easily

solved.[3] Previously, debuggers that tried to solve this were unacceptably slow as they machine-stepped (slowly) through the non-debug routines until they detected a new routine that had debug information. The performance penalty of this approach is so extreme as to make any debugger employing it laughable.

There are two workable solutions to this problem. One is to build into the stepping algorithms a "run to first source" feature. To implement this, a breakpoint is placed on all procedure entry points. This way, stepping off the end of a function or into code with no source will stop the next time a user-written function is entered. This solution requires very fast lookup, setting and unsetting of a large number of function entry point breakpoints. The alternative is to use page protection set for all user-written code pages that will cause a page protection violation as soon as user code is about to execute again.

If the OS provides APIs to control the memory access permissions on a page-by-page (in the OS sense) basis, these can be used to set the code pages of the debuggee corresponding to the portion having debug information as not executable and not readable. When step into lets the debuggee run full speed to the internal breakpoint set at the non-debuggable function's return address, if user code gets called before that internal breakpoint is hit, a page protection violation exception will cause the debuggee to stop. The debugger will then be able to use this exception that stopped the debuggee to reset the page protection and to continue stepping as before into the user's "hidden" routine as desired. This is a major advancement for debuggers because this stepping problem has plagued all debuggers to date. Algorithm 6.6 shows this modification to the standard step into algorithm.

C++ Global Constructors and Destructors

A similar flaw in source step occurs at initial startup and on final shutdown of a C++ application. As a C++ program starts up, after initial loading into memory and execution of the run-time startup code, all C++ global constructors must be executed *before* the function 'main' begins execution. However, most debuggers run to main and stop there, presenting the illusion to the user that program execution actually begins at 'main.' The goal was to run to the first user code, which was most easily accomplished by running to 'main.'

[3]Currently, I know of only Borland's Delphi and C++ debuggers as ones that specifically address this issue.

Algorithm 6.6 *Step into "hidden" debuggable routines*

Method This is a modification to Algorithm 6.4.

i. Perform all steps of the step into algorithm up to and including setting the breakpoint on the return address of the first non-debug routine found;
ii. Before letting the debuggee run full speed do the following:
iii. Set all code pages of debuggee as non-executable and/or non-readable using the memory access API of the operating system;
iv. Run the debuggee full speed;
v. If (internal function return address breakpoint hit) then
continue normal processing of step into algorithm;
else
set all debuggee code pages back to executable
go back to step i of algorithm 6.4

This was reasonable for C-language debuggers, but in C++ the critical bug may be in one of the constructors that execute before main. Therefore, running to main does not yield the expected or desired result. On the other hand, machine-level stepping will be too slow to be usable to avoid this problem. Here at least, unlike the previous scenario, it is possible to get help from the run-time library as to where the constructor chain begins. The global destructors executed after main returns present the same situation. (In fact, bugs here are quite common as a user begins to track down memory leak bugs.) The same technique used for hidden routines works here but it critically depends on a page protection API provided by the OS.

Step-related Algorithms

There are a series of features a debugger may provide that superficially do not appear directly related to stepping algorithms. But, in fact, a wide range of these features do directly utilize the stepping algorithm. The list we will briefly consider includes the following:

- Animation
- Software watchpoints
- Finish function
- Reverse execution

- 'Slime' trail mode
- C++ exceptions

Animation

Animation is used to dynamically show the progress of execution through the program. It is sort of a "watch the bouncing ball" for a computer program. It is implemented by continuously executing the step-into algorithm pausing at the end of each statement step to refresh the views but then immediately resuming stepping. This is a marginally useful feature, usually used for demonstrations, program learning, or testing. A debugger supporting a scripting or macro capability—a very useful feature—can easily overlay animation on the standard step-into feature without engineering in animation.

Software Watchpoints

Software watchpoints are data access breakpoints that are not implemented via hardware assist. A watchpoint "watches" a range of data addresses in debuggee memory and activates—stopping the debuggee—if any modification is attempted. For memory corruption bugs this is a critical feature. There may be no hardware assist or the limit on the number of locations watchable has been exceeded which causes the need for software implementation. One approach to implementation in software is to invisibly, even on a "run" request, use the single step algorithm checking the specified address ranges for change on each step completion. Totally accurate results actually require using machine-step granularity, but in practice this is much too slow. Even using source step granularity is so slow that clear warning about slow stepping to the user is prudent.

Alternative implementation strategies may be more effective, depending on the OS facilities provided. For example, if the debugger can mark a page of debuggee data memory as read-only, then an attempt to write into that page would cause an exception that the debugger would catch. On this type of exception the debugger would check the exact address of the access attempt, and if it intersects a watchpoint, activate that watchpoint. Otherwise, the page must be made writable, the process stepped a bit, and then re-marked read-only before a full-speed run can be resumed.

Finish Function

A feature that can be thought of as "finish the current function" is very useful at times. Once a user finds that he or she has stepped into a function

inadvertently he or she might like to run quickly to a point just after the call to this function. There are two ways to offer this kind of functionality. One is to require users to use the call stack to select one stack frame prior to the current function. This will show the place from where the current function was called. Here a breakpoint or "run-to-here" could be performed. Shorthand for this—and more convenient to the user—would be an explicit "finish function." The implementation of this is just the latter half of the step-over algorithm. The result is to very quickly take the user to the source line immediately succeeding the call to the current function.

Reverse Execution

Truly effective reverse execution would be a very valuable feature for a debugger. Some interpretive language systems can perform true reverse execution because they have complete control over all program states. Debuggers in compiled language systems must settle for a very limited form at best. When a bug is observed it would be very valuable to "back up" a little to examine program state thoroughly just before the fault occurs. This would allow the user to zero in very quickly on the cause of a fault. In a compiled language system this would require noting all memory accesses and saving memory state prior to any changes so these can be reversed. This requires instruction single-stepping and decoding all instructions for their reversible state. Even so, many instructions are simply not reversible, such as I/O instruction or any calls to OS or library routines. Because it requires special processing during single-step, reverse execution requires a mode set by the user. Only when this mode is set does the debugger single-step and save the state of each memory location about to be modified. These limitations make reverse execution a practically useless feature. The goal of backing up from where the fault occurred can almost never be met. In spite of its limitations, users still request the feature in debuggers, and some commercial debuggers (for example, Turbo Debugger) have attempted to implement it.

Slime Trail

Because "where am I" and "how did I get here" are perhaps the most frequent debugging questions, a debugger should be prepared to answer them. "Where am I" is covered by source views, disassembly views, and a stack back-trace. But "how did I get here" is not completely answered by a stack trace. Functions entered but already left do not appear on a stack back-trace. For this reason, some debuggers have a "slime trail" mode that shows all statements and functions executed up to the current execution point. This is

usually done just like animation using single-step—instead of showing the user each step, a record is emitted for each statement that is viewable at a later time by the user.

C++ Exceptions

C++ exceptions consist of a "throw" at the point of detection and a "catch" of the prescribed type at some point programmed to handle the fault more cleanly. The most important benefit of C++ exceptions is that the compiler guarantees all destructors for all automatic objects will get called on the way up the stack from the throw to the catch. Defects can occur anywhere including in the throw logic, catch logic, or any destructors in between. Thus, the debugger must aid in tracing through this code. An implementation strategy for C++ exceptions is to have the debugger aware of a single, well-specified run-time library special dispatch routine where all C++ throws are initiated. From here, users can be given the option of running immediately to the catch point or stepping through each destructor in succession. In this latter case, the step algorithm is used once again to trace through the destructors as if they were a series of nested routines.

Event-driven Stepping Models

The standard stepping model presented so far assumes a user builds programs by building up a series of routines, one directly calling another. Another model occurs frequently in strictly event-driven systems such as that provided by Visual Basic and Delphi. Each user function represents an action taken on some system-specified event. The user associates one function with each event. But these user functions are only directly called from system dispatch routines (which will not have debugging information and will thus be "invisible" to debuggers). Our standard stepping algorithm will fail completely on this model. Even if a stop at a breakpoint occurred in one of the user's routines, stepping off the end of this function will end up back in system code that was responsible for calling the user function. The simplest algorithm for handling this type of programming model, and the one employed in Visual Basic and Delphi, is to have the debugger detect when it is about to step off the end of a user function and at that moment set breakpoints at the beginning of every user routine in the entire system. Once the user routine returns, the debugger must cause the debuggee to run at full speed. Eventually, the debuggee will hit one of the special breakpoints and stop at the start of another user routine. We specify this in Algorithm 6.7.

Algorithm 6.7 *Event-driven stepping model*

Input Current statement and instruction pointer address.
Output A new statement and instruction pointer address for the 'logical' next statement.
Method The key is detecting stepping off the end of a user routine and then catching the next time execution moves back into user code.

i. Detect step off end of user routine;
ii. Get list of all user routine entry points from symbol tables;
iii. Set internal breakpoint at each entry point (or, if memory protection is available, mark all debug information code pages as inaccessible);
iv. Run debuggee full speed;
v. Stop at special breakpoint will occur some time later;
vi. Remove all special breakpoints;
vii. Revert to standard stepping algorithm.

I should note that one of the unsolved stepping problems mentioned earlier is closely related to this stepping model issue. The problem of stepping from user routine to system routine that calls another user routine is identical to this VB model. However, in standard programming models, setting breakpoints on each and every user routine whenever single-step needs to run full speed to a breakpoint is impractical. Unfortunately, some other solution must be employed to address this problem.

chapter 8

Inspecting Data and Variables

The execution control algorithms we have just described perform half of the task of debugging: letting the user run and stop the program at will and inspect debuggee context via source, stack, and CPU information. Once the program has stopped, data inspection algorithms perform the other half of the task, letting the user examine and alter the program's data structures. Execution control lets the user understand the "where" of a bug; data inspection lets the user understand the "how."

Evaluating Expressions

Ideally, a symbolic debugger can display data by evaluating an expression that uses the same identifiers and syntax that appear in the source program. This requires the debugger to implement an interpreter for the expression syntax and semantics of one (or, in the case of a multilanguage debugger, more than one) source language.

Many issues for the debugger's interpreter are no different from those covered by the literature on conventional interpreters. The primary difference is that the debugger's interpreter does not allocate its own storage for variables, but instead accesses them within the debugger's child process at the addresses specified by the debugging tables emitted by the compiler. This is critical because what the user is after is the use of the actual values being used by the running program.

One obvious implementation is to adapt the parsing and semantics phases of the compiler that emits the programs on which the debugger must operate. The adapted compiler front-end reads the text typed by the user of the debugger and builds its parse tree as usual. It consults the debugging tables to obtain the identifier and data type information, which in the original compiler would have been placed in the compiler's own symbol table by declaration statements. Then it performs its usual semantic checking and annotation. Next, a new piece of debugger-specific code walks the tree, replacing variables with values obtained from the child process. Finally, the usual constant-folding code within the compiler combines the values and delivers a result.

This approach saves some duplication of work, and it encourages the compiler and the debugger to behave alike; in particular, the debugger can even deliver the same error messages that the programmer is accustomed to receiving from the compiler. The most important advantage to this approach is that *the debugger evaluator is now guaranteed to use the same language syntax and semantics as used in the construction of the underlying program*. The alternative is to build an expression evaluator and language parser as a special dedicated piece of code in the debugger.

There are significant differences between the compiler's needs and the debugger's. If possible, anticipating the debugger's needs when designing the compiler will minimize difficulties.

For example, whereas a compiler always evaluates an expression in the context of the current scope, a debugger may let the user point to any frame on the stack and evaluate an expression as if that were the topmost frame. At best this simply requires passing an extra parameter to the existing compiler code, which maps identifiers onto symbolic information; at worst it requires considerable new work. For example, one simple compiler symbol table organization allocates a block of symbol table entries on a stack at the beginning of each new scope and deallocates the block at the end of the scope. The algorithm for searching the scoping hierarchy is implicit because relevant scopes appear on the stack in the proper order and irrelevant ones have been discarded. This scheme is not sufficient for a debugger, however, because all scopes are present in the debugging tables all the time; the debugger must therefore explicitly choose which scopes to search.

Even within a particular scope, the debugging tables are unlikely to use the same data structures as the normal compiler symbol table. One good solution to this problem is to provide pairs of symbol-table access methods, one for the compiler and the other to translate the object file debugging tables into the data structures the compiler expects to see.

Such translation may need to operate in a "lazy" or "as-needed" fashion. Consider an expression that refers to a single data member of a C++ class. A complete translation of the class would need to process its base classes and then each of those classes' base classes in turn. If the class hierarchy is highly intertwined, this could ultimately require translating every user-defined type in the entire program.

If the debugger supports more than one language, additional problems arise because the user can attempt to evaluate an expression that combines operators from one language with operands generated by different languages, leading to undefined results. An easy solution is to tag each symbol with its source language and to refuse to mix languages within a single expression, but the user may consider this draconian: such a debugger would, for example, refuse to copy a value from a Pascal global integer to a C global int. A better solution is to build checks into the code that translates data structures from the debugging tables into data structures within the compiler symbol table. Gross mismatches are easy to exclude: for example, one cannot map a C pointer onto any FORTRAN 77 data type. Subtle matches still require care: for example, although it might seem that a Pascal tagged variant record maps easily onto a C union embedded within a structure, that might not be true if the C semantics phase assumes that such a union will have a name (because the C parser would not permit its omission), and the absence of a name causes a bug. Or consider the problems that might result if a C++ compiler, which assumes that any symbol named "this" will have a structured data type, encounters a Pascal scalar global named "this."

Adapting the compiler for use within the debugger may also require extra work in constant folding; whereas a compiler-writer can decide that it isn't profitable to fold a difficult and rare combination of operator and operand types, the user will complain if this causes the debugger to reject a legal expression.

Scope Resolution

If a debugger is meant to work with more than one language, it must avoid the temptation to build into its symbol-access methods any assumptions about identifier scoping because these differ among languages. Instead, each language-specific evaluator must bind identifiers onto data items using the rules for the language. For example, a Pascal evaluator that fails to find an identifier in the scope of the current function will next search the statically enclosing function; a C evaluator will next search the scope of the file; and a C++ evaluator will next search for a data member belonging to the same class as the function.

It is useful to extend the syntax of each language while in the debugger's evaluator subsystem to let the user specify variables that exist but that are not currently in scope. For example, the special inspector syntax "#gcd#i" might specify the variable "i" in function "gcd", letting the user examine this variable whenever a frame for "gcd" exists on the stack. Or "##i" might permit the user to examine a global variable named "i" even though the current function redeclares the name "i" as a local variable.

If you allow the user to set a watchpoint (a data breakpoint) on a local variable in procedure X, which is stack allocated, you must somehow ensure that the watchpoint does not fire on some other variable that happens to use the same memory when procedure X is not active. Either the debugger must automatically disable the watchpoint when procedure X returns and reenable it on the next call to procedure X, or it must ignore spurious firing of the watchpoint when procedure X is inactive. It must also deal with the possibility that procedure X may be recursive.

A home table is a list of program-counter ranges where each range specifies a variable's location when the program is executing within that range. If the debugger uses home tables to track when the compiler moves variables into and out of registers, it may need to deal with the fact that the variable spends part of its time in a register instead of memory. There is no hardware or operating system support for an exception if a register's value is modified so the watchpoint will not fire and the debugger will not have been truthful.

Automatic Redisplay of Expressions

Some debuggers will automatically reevaluate an expression each time the child process stops. This feature (sometimes called an "inspector") can require extra work to achieve both correctness and good performance.

First, the evaluator must separate the mapping of identifiers onto symbols from the rest of the evaluation work. This ensures that each identifier has the same meaning each time the debugger reevaluates the expression; it also reduces the cost of reevaluation because only the first evaluation must search scopes to resolve identifiers. Second, it must provide a list of the scopes required, so that the debugger avoids reevaluating an expression unless the stack contains a frame for each of those scopes.

Invoking Functions during Evaluation

The ability to invoke a function during expression evaluation is important in languages like C++ because the user may inadvertently use an operator in an expression that has been overloaded with a function. Following is a sample fragment of C++ code showing function overloading that makes a simple operator actually turn into a function call.

```
operator+ cadd(int,int);
c = a + b; // this '+' calls function cadd()
```

Although the debugger can interpret most expressions, as opposed to generating native code, an expression that invokes a function poses a problem. It is generally not practical to parse the entire function, process it semantically, and then use the resulting tree to drive an interpreter. Even if all this was done, the debugger's expression "language" must remain completely "bug compatible" with the compiler, and this is virtually and practically impossible.

Thus, most debuggers that permit expressions containing function calls employ a trick. Instead of interpreting the function invocation, the debugger builds an argument list using the stack within the child process, sets a breakpoint at the function return address, and starts the child process running at the beginning of the function. When the child process reaches the breakpoint,

Algorithm 8.1 *Invoking a function during expression evaluation*

Input Function and actual argument list
Output Function return value, plus side effects
Method Use the child process to execute the desired function

i. Evaluate each of the actual argument expressions and save the resulting value in the debugger address space. (Remember that any of these expressions may itself invoke a function.)
ii. Save the child process registers and program counter.
iii. According to the calling conventions of the target machine and compiler, push onto the stack (or copy into a register) each of the actual argument values. (For example, the rules for a non-scalar return value may require you to allocate space on the stack prior to pushing the arguments.) If the language allows user-written exceptions, set up the necessary machinery so that if the function throws an exception, the run-time system will not unwind past this point without giving control back to the debugger.
iv. Choose a "distinctive" return address. Set a breakpoint at that address. As dictated by the calling conventions, push the address onto the stack or copy it into a register.
v. Copy the starting address of the function into the program counter register.
vi. Run the child process.
vii. When the child encounters the breakpoint, retrieve the return value according to the calling conventions (a non-scalar return value may lie within the space mentioned in step iii.)
viii. Remove the breakpoint set in step iv.
ix. Restore the registers and program counter saved in step ii.

the debugger retrieves its return value and restores the child process stack and registers to their original state.

The trick assumes that it is safe to invoke a function at a point where no function invocation appeared in the original program. Fortunately, the assumption is valid for most code generators today. (Given a sophisticated optimizer, this can be invalid because it is equivalent to inserting the function invocation at the current point in the program; even the techniques for debugging optimized code fail to address this because they merely describe the original program graph to the debugger.) As a concrete example, consider

a function that makes up-level references to a variable that is dead or enregistered at the current point in the program.

The return address used in this algorithm should be "distinctive" enough so that neither recursive nor non-recursive calls to additional functions will erroneously trigger the breakpoint. The debugger must also anticipate that the function may not return: It may encounter a breakpoint set by the user, or may fault, or may terminate execution of the child process, or may stop executing via a non-local goto or C language "longjmp" that bypasses the normal return address.

An easy solution is to disable user-set breakpoints temporarily and to treat a fault as an error that reports failure to the user and restores the child process state. Process termination or non-local goto constitutes a more serious error because it may be impossible to restore the state. I should emphasize that encountering breakpoints during function evaluation, while it might add on interesting set of capabilities to the user, is quite dangerous if not handled very carefully in the debugger. The most significant issue is reentrancy. The debugger itself has recorded a stop and recorded critical debuggee state information. Now a disjointed path of execution is being followed where another stop is encountered. What is shown on the stack? What are the other threads of execution doing? If they are not frozen they just got to run and create unforeseen side effects. In general, I'd suggest the return on investment for this capability is very low: don't allow breakpoints during function evaluation.

Compiler-generated Debugging Information

A symbolic debugger depends on the compiler and linker to emit debugging tables (often called a "symbol table," but not to be confused with the symbol table used within the compiler itself) that describe the mapping from names and statements within the source program onto the object program (MICROSOFT 1993).[1] But a compiler usually emits the object program and the debugging symbol table separately, so an error in the symbol table appears to the user to be a "bug" in the debugger.[2]

---

[1]Nearly all symbol table encodings are proprietary, an issue which makes it difficult for vendor X to handle vendor Y's encoding and for the encoding to improve through open review.

[2]In fact, as a user of a debugger, many of the bugs you have encountered have almost certainly been due to compiler-debug information problems, operating system bugs, or other issues not directly controlled by the debugger developer.

Ideally (as explained later in connection with the question of debugging optimized code) the debugger might do better to access the same intermediate data structures that the compiler uses to represent the program, but that approach is not common practice due to problems with bulk and information hiding. Instead, the debugging symbol table provides only the information the debugger is thought to need. Often the author of the debugger has no voice in the design of the symbol table, and some debuggers must cope with a variety of formats, typically by translating them into an internal form.

A debugging symbol table must deal with a number of issues:

- Does the symbol table cater to the compiler or to the debugger?
- How does it divide the work among compiler, linker, and debugger?
- Does it permit incremental processing and caching of information?
- Can it support a variety of target machines?
- Can it support a variety of compilers?

Catering to the Debugger

The symbol table is a database, and as with any database, the best organization depends on which queries need to be fast. Unfortunately, queries that are important to the debugger may not occur at all within the compiler. For example, both the debugger and the compiler query variables by name when they process expressions; only the debugger queries them by memory address (for example, when it disassembles a memory-referencing instruction and wishes to print the identifier corresponding to the operand in memory).

The needs of the compiler and debugger also differ because the compiler deals with one compilation at a time, whereas the debugger deals with the entire executable. The debugger may, for example, be confronted with a much larger number of global variables; it may need to acquire (and later discard) additional symbolic information during execution of a program that relies on dynamically linked libraries. We have found that most debuggers have capacity problems due to these issues that must constantly be addressed by the debugger developers.

Dividing the Work

Clearly someone—the linker, on one hand, or the debugger, on the other—must reorganize the data to suit the needs of the debugger.

It might seem best to make the debugger perform all the work: Although the first invocation of the debugger might be slow, the debugger could cache the reorganized data in case it is invoked again before recompilation. Compilation can be fast, and the linker need incur no extra expense at all, provided the debugger can retrieve individual symbol tables from the relocatable object files and resolve relocations itself (a task that it must perform for dynamically linked libraries anyway). Best of all, the debugger may be able to avoid processing some relocatables if the user doesn't refer to them during the debugging session.

Most systems take the opposite approach, however: Linkers combine symbol tables from the relocatable object files and perform at least some reorganization before writing the information to the executable file.

There are several reasons for this:

- Programmers may wish to delete relocatables after linking.
- Binding the symbol table to the executable reduces the chance of losing or mismatching the symbol table.
- The debugger can avoid duplicating work (such as organizing public symbols for rapid access by name) that the linker must perform anyway.
- The linker may reduce the volume of the symbol table dramatically by eliminating duplications.

The last point is probably the strongest argument for involving the linker. The data types, global variables, and procedure definitions that make up the interfaces between separate compilations are defined once by the exporter of the interface and referenced repeatedly by the importers. In a language like Modula-2 - an early object-oriented language used frequently in Europe—which describes the interface via a definition module, it is easy to represent the interface once for the exporter without repeating it for any of the importers. But for languages like C and C++, which rely on textual inclusion, symbol tables can grow explosively. For example, consider that dozens of compilations in a large program may include the same file "stdio.h" or "iostream.h."

Some systems attempt to treat the ".h" file like the Modula-2 definition module; the compiler segregates symbol table information generated by the ".h" file and the linker discards duplicate copies. The linker must ensure that the copies are truly identical, however, because the programmer may have legally used conditional compilation so that the same ".h" file generates different information in different compilations. Other systems require the linker or a post processor to hash or sort all symbols and types in the program to eliminate duplicates (MICROSOFT 1993). These horrendous complications that plague C++—due to header files and executable code in these headers—is one of the major motivating factors behind the design of Java.

Incremental Processing

Reading and processing the entire symbol table for a large program can cause an annoying delay. Fortunately, a typical debugging session exhibits a great deal of locality (LINTON 1986). If the debugger can read the symbol table in a lazy or incremental fashion, it may entirely avoid reading most of the symbol table, and can divide the remaining processing into a number of smaller, less noticeable, delays. Linton's studies showed that most debugging sessions required less than 15 percent of the available symbol table was needed.

Different Target Machines

Obviously different target machines will differ with respect to word size, address range, registers, the use of segments, and so on. Less obviously, there is a trade-off between minimizing the size of the symbol table and supporting a variety of targets. For example, if a target machine has a hardware protocol for saving registers in the prologue of a procedure, or if the debugger can easily infer the identity of saved registers by disassembling code, the symbol table need not list them. But a symbol table that cannot represent the names and locations of saved registers may be unusable on a machine whose protocol is complicated.

Compilers vary in the way they build runtime structures such as dope vectors or C++ virtual function tables. Obviously, different compilers will require different information, and again there is a trade-off between size and generality.

Accessing Symbol Tables

How the debugger accesses the symbol table depends, of course, on which services the debugger provides to the user, but most debuggers have a certain set of queries in common. The following list describes each query and gives one or more examples of its use.

1. *Map instruction address onto the enclosing scope.*
   When the user asks to evaluate an expression, the debugger must use the appropriate scope. When the user asks to trace the stack, the debugger must show the scope or procedure corresponding to each return address on the stack.
2. *Map scope onto statically enclosing parent scope.*
   For statically scoped languages, evaluating an expression often requires searching a hierarchy of scopes. When tracing the stack, the debugger may prefer to show the innermost enclosing function in place of an unnamed lexical scope.
3. *Map scope plus identifier onto type and location.*
   When evaluating an expression, the debugger must search for an identifier within a scope, and then use its type plus its location (which may be a memory address, a register, a constant value, or some combination) to fetch from the child process the number of bits indicated by the data type. This query will need to find functions as well as data and must handle global scope as a special case.
4. *Map instruction address onto source statement.*
   When the program stops, the debugger must show the current source statement.
5. *Map code or data address onto statically allocated variable or procedure.*
   When the debugger disassembles instructions, it may wish to show the names of the variables and procedures to which they refer.
6. *Map source statement onto instruction address range.*
   When the user sets a breakpoint on a source statement, the debugger must find the first instruction of that statement. When the user asks to step through a source statement, the debugger must find the end of the statement.

A Sample Symbol Table: STI

STI, also known as CodeView debug format, illustrates a typical debugging symbol table. It is complicated somewhat by its need to support segmented 16-bit and 32-bit addresses for Intel x86 machines as well as simple 32-bit addresses for non-Intel target machines. It is complicated further because the compiler emits one set of tables, the linker combines these and rewrites them into a different set, and a post-processor called CVPACK transforms that into yet a third set. In the following discussion I'll ignore some of the details required solely for 16-bit machines or for segmentation, I'll concentrate on the third set of tables, and I'll omit some less important details entirely.

The compiler uses two kinds of records to describe most program objects:

$$SYMBOLS	Descriptions of procedures, variables, named constants, and named types
$$TYPES	Descriptions of scalar, array, aggregate, and enumerated types

The $$SYMBOLS section is a series of variable-length records. Each begins with a 2-byte length field, followed by a 2-byte opcode field that indicates the purpose of the record and dictates the set of fields that occupies the remainder of the record.

For example, a record whose opcode is S_GDATA32 describes a 32-bit-addressed global variable and contains the fields listed in Table 8.1.

As a second example, a record whose opcode is S_BPREL32 describes a 32-bit-addressed local variable (Microsoft compilers address these relative to the base pointer register of the Intel 80X86 machine, hence the opcode name), as listed in Table 8.2.

Other record formats include the following:

S_REGISTER	Register variable
S_CONST	Constant
S_UDT	User-defined type
S_LDATA32	C "static" variable
S_LPROC32	C "static" procedure
S_GPROC32	Global procedure
S_THUNK32	Thunk procedure

TABLE 8.1 *Encoding for 32-bit addressed Global Variable*

FIELD	SIZE IN BYTES	PURPOSE
length	2	Number of bytes in the record, excluding the length field itself
S_GDATA32	2	Opcode describes symbol, dictates which additional fields follow
offset	4	Address of the variable (offset part)
segment	2	Address of the variable (segment part)
type	2	Index of the data type within $$TYPES
name	variable	Variable name (length-prefixed string)

TABLE 8.2 *Encoding for 32-bit addressed Local Variable*

FIELD	SIZE IN BYTES	PURPOSE
length	2	Same as above
S_BPREL32	2	
offset	4	Signed offset from BP register
type	2	Index of the data type within $$TYPES
name	variable	Variable name (length-prefixed string)

S_BLOCK32	Nested lexical scope
S_WITH32	Pascal "with" statement
S_END	End of scope of procedure, lexical scope, or "with" statement
S_LABEL32	Statement label
S_VFTPATH32	C++ virtual function table path descriptor

Records describing procedures, nested lexical scopes, and "with" statements are threaded together to describe the scoping structure of the program. In each of these records, one field points to the parent scope and another field points to the next sibling scope within that parent's scope. All records belonging to a scope must appear immediately after the record for the scope itself. A third field within that record points to the last record belonging to it. A record called S_SSEARCH, which must appear at the beginning of the $$SYMBOLS section, points to the procedure at the head of the list.

For example, Figure 8.1 shows the $$SYMBOLS records for the following compilation:

```
procedure outer;
    var outer_var0, outer_var1: integer;
    procedure inner0;
        var inner_var: integer;
        begin
        end;
    procedure inner1;
        begin
        end;
begin
end;
```

The structure of individual records within the $$TYPES section is similar to that of the $$SYMBOLS section, except that one record may contain a series of leaf structures, each structure having one opcode and a variable number of fields dictated by the opcode. The 2-byte length at the beginning of the record counts the number of bytes in all the leaf structures.

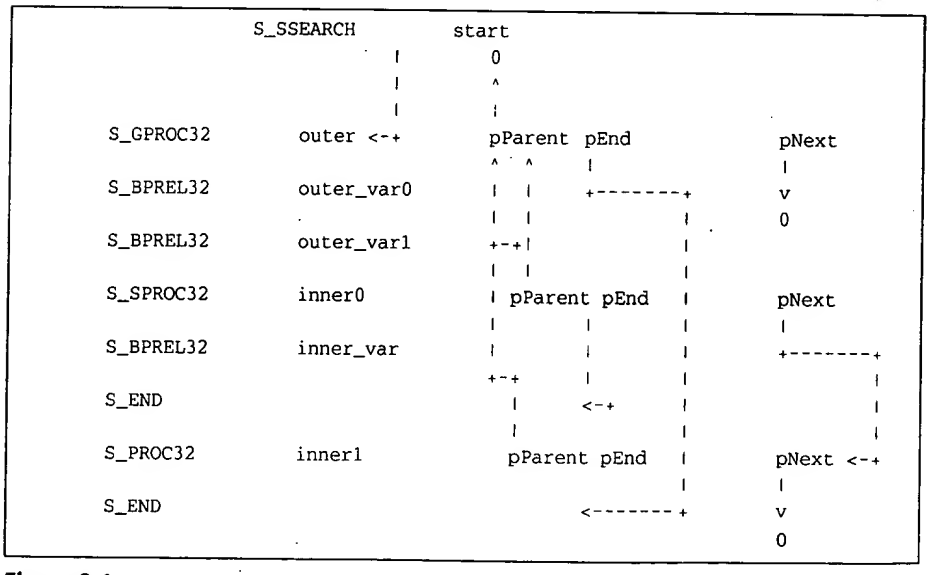


Figure 8.1

Layout of $$SYMBOLS records in STI graphically. *The layout and connections of the $$SYMBOLS records for the compilation of a simple pascal procedure and two nested inner procedures.*

For example, this record describes an array with default lower bound and constant upper bound, as shown in Table 8.3.

As a second example (Table 8.4), this record describes a C++ class.

Other opcodes for type records include the following:

LF_POINTER	Pointer to type
LF_ENUM	Enumerated type
LF_PROCEDURE	Procedure type
LF_METHODLIST	List of C++ member functions
LF_FIELDLIST	List of members of C or C++ struct, union, or class
LF_BITFIELD	C bitfield member
LF_ARGLIST	List of formal arguments
LF_VFUNCTAB	C++ virtual function table

Each record is assigned a number beginning at hexadecimal 1000. When a field points to another record, it uses this number. Numbers below 1000 are reserved for various intrinsic scalar types (such as integer or double-precision real).

STI requires a substantial amount of processing by the linker and the CVPACK post-processor. Some of it makes access more convenient for the debugger. For example, because type records vary in length and use record numbers rather than byte offsets to point to one another, the debugger would have difficulty following these pointers. So CVPACK creates an array that maps type numbers onto offsets relative to the beginning of $$TYPES. In addition, CVPACK separates global symbols from the rest of the symbols and puts them in a separate table, optionally creating hash tables to reduce the cost of searching the globals by name or by address.

Other post-processing is mandated by the design. For example, CVPACK must eliminate redundant $$TYPES records generated by separate compilations because the address of a C++ class method lies in a record separate from the description of the C++ class itself, and it is not generally possible for the debugger to recognize the association between these records unless each data type has a unique index.

When the linker and CVPACK are finished, the executable file has one set of the following tables for each relocatable object:

TABLE 8.3 *Encoding for an array with the Default Lower Bound and Constant Upper Bound*

FIELD	SIZE IN BYTES	PURPOSE
length	2	Number of bytes in the record, excluding the length field itself
LF_DIMCONU	2	Opcode describes type, dictates which additional fields follow
rank	2	Number of dimensions
index	2	Points to record describing the data type of array index
bound	rank * s	Constants specifying upper bound of each dimension; "s" is the number of bytes each constant occupies, dictated by the index type

TABLE 8.4 *Encoding for a C++ Class*

FIELD	SIZE IN BYTES	PURPOSE
length		Same as above
LF_CLASS		
count	2	Number of members
memberlist	2	Pointer to another record listing the members
property	2	Bit mask describing attributes of the class (for example, whether it is packed, whether it has overloaded operators, etc.)
dlist	2	Points to record describing the classes that inherit this class
vshape	2	Points to record describing the virtual function table
length	variable	Size in bytes of the class
name	variable	Class name

sstModule	Address ranges of code and data emitted
sstAlignSym	$$SYMBOLS records for non-globals
sstSrcModule	Mapping from source statements to instructions

The executable has exactly one set of the following tables:

sstGlobalTypes	$$TYPES records for all types
sstGlobalPub	$$SYMBOLS records for public data
sstGlobalSym	$$SYMBOLS records for global procedures

The sstSrcModule table consists of a header followed by an assortment of file-information records and line-information records. The table contains these fields:

cFile	Number of source files contributing code to this compilation
cSeg	Number of segments receiving code from this compilation
baseSrcFile	Array [cFile] of pointers to file information records
start/end	Array [cSeg] of pairs of offsets (giving the range of addresses for each segment)
seg	Array [cSeg] of segment indices, corresponding to the start/end pairs

file-information record for file 0

line-information records for file 0

...

file-information record for file n

line-information records for file n

Each file information record contains these fields:

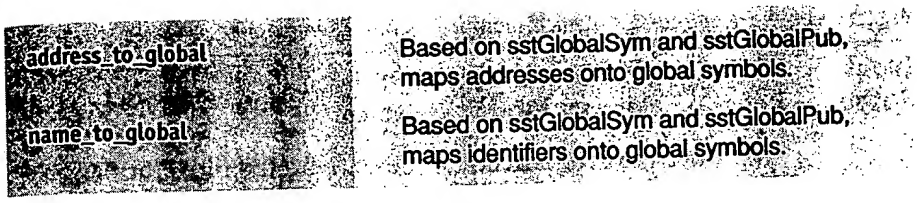
cSeg	Number of segments receiving code from this file
baseSrcLn	Array [cSeg] of pointers to line information records, one per segment
start/end	Array [cSeg] of pairs of offsets (the range of addresses for each segment)
Name	Length-prefixed file name

Each line information record associates an array of line numbers with a parallel array of segment offsets. It contains these fields:

Seg	Segment index
cPair	Number of source lines

TABLE 8.5 *Basic Data Structures Built by Debugger for Access to Symbolic Information*

DATA STRUCTURE BUILT	DATA STRUCTURE DESCRIPTION
module_map	Maps any address onto the corresponding module, indicating whether to obtain the symbols for that module from the file or from module_symbols.
module_symbols	For each module, represents the corresponding sstAlignSym information.
type_vector	For each data type index, points to the corresponding record either in the file or in type_records.
type_records	Types records that have been read from the file.
global	Special, reserved pointer indicating global scope.
file_to_module	For each file, gives the set of modules containing code generated from that file. We assume that a particular file is usually associated with only one module, but to handle the more general case we are willing to iterate through a list of modules.
address_to_statement	For each module, provides an array of (address, file, statement) tuples, sorted by address.
statement_to_address	For each module, provides an array of (statement, pointer) tuples, sorted by statement, where each pointer indicates a tuple in the address_to_statement entry for that module. We assume that within a module, all statements usually lie within the same source file, but to handle the general case we are willing to iterate through a list of tuples, selecting the one having the appropriate file. Because the STI format provides only the starting address for a statement, we must infer its ending address from the starting address of the next statement in the symbol table. Thus, instead of storing addresses within this data structure, we store pointers into the address_to_statement data structure, where the addresses appear in order.



x. If we have not already done so, read the type record from sstGlobalTypes and add an entry to type_records.
xi. Use the location fields within this symbol record to determine the location and return success.
xii. If this record is a procedure, scope, or "with" statement, advance past its "pEnd" record; otherwise, merely advance to the next record.
xiii. Go to step vi.

When a breakpoint fires and whenever we need to map an instruction address onto the correct source statement, Algorithm 8.5 is needed. This algorithm takes the instruction address and, using source line information contained in sstModules, determines the correct file name and line number.

The opposite mapping—from source statement onto instruction address range—is used by the source view to show the breakpointable lines. It is also used whenever the user sets a breakpoint on a source statement. This mapping is shown in Algorithm 8.6.

Algorithm 8.5 *Map instruction address onto source statement*

Input Instruction address
Output Source statement filename and line number

i. If we have not already done so, read sstModules and construct module_map.
ii. Use the input address plus module_map to find the appropriate module.
iii. If we have not already done so, read sstSrcModule for that module and construct address_to_statement and statement_to_address entries for that module.
iv. Search address_to_statement for the highest address that does not exceed the input address.

Algorithm 8.6 *Map source statement onto instruction address range*

Input File name and line number
Output Range of addresses for that statement

i. If we have not already done so, read the file information records from each sstSrcModule and construct file_to_module.
ii. Use the file name plus file_to_module to select each module that might contain the desired statement.
iii. If there are no (more) candidate modules, report failure.
iv. If we have not already done so, read the sstSrcModule for the candidate and construct statement_to_address and address_to_statement.
v. Use the line number along with statement_to_address to select all of the tuples having the desired line number. If the set is empty, advance to the next candidate module and go to step iii.
vi. For each selected tuple, follow its pointer to the corresponding tuple within the address_to_statement table, and compare the file name with our input file name. If no tuple matches, advance to the next candidate module and go to step iii.
vii. Use the offset from the address_to_statement tuple as the low bound of the range. Return the offset of the next tuple in address_to_statement as the high bound of the range.

Statically allocated variables have a simple mapping from a data address. Statically allocated procedures have the same mapping, shown in Algorithm 8.7.

Algorithm 8.7 *Map code or data address onto statically allocated variable or procedure*

Input Address
Output Pointer to record in module_symbols

i. If we have not already done so, read sstGlobalSym and sstGlobalPub and construct address_to_global and name_to_global.
ii. Search address_to_global for the highest address that does not exceed the input address.
iii. Use the data type of the selected symbol to determine the range of addresses it covers. If the input address does not lie within that range, return failure.